Generalization and Abstraction

We use the terms **Abstraction** and **Generalization**, when we start trying to model real world things in software.

Before I launch into interface types and abstract classes, I want to talk about what these concepts mean.

Generalization

When you start modeling objects for your application, you start by identifying what features and behavior your objects have in common.

We generalize when we create a class hierarchy.

A base class is the most general class, the most basic building block, which everything can be said to have in common.

Abstraction

Part of generalizing is using abstraction.

You can generalize a set of characteristics and behavior into an abstract type.

If you consider an octopus, a dog, and a penguin, you would probably say they're all animals.

An animal is really an abstract concept.

An animal doesn't really exist, except as a way to describe a set of more specific things.

If you can't draw it on a piece of paper, it's probably abstract.

Abstraction simplifies the view of a set of items' traits and behavior, so we can talk about them as a group, as well as generalize their functionality.

Java's support for Abstraction

Java supports abstraction in several different ways.

- Java allows us to create a class hierarchy, where the top of the hierarchy, the base class, is usually an abstract concept, whether it's an abstract class or not.
- Java let's us create abstract classes.
- Java gives us a way to create interfaces.

Abstract method

In the videos that follow, I'll be talking a lot about abstract and concrete methods.

An abstract method has a method signature, and a return type, but doesn't have a method body.

Because of this, we say an abstract method is **unimplemented**.

It's purpose is to describe behavior, which any object of that type will always have.

Conceptually, we can understand behaviors like move or eat on an Animal, so we might include those as abstract methods, on an abstract type.

You can think of an abstract method as a contract.

This contract promises that all subtypes will provide the promised functionality, with the agreed upon name and arguments.

Concrete method

A concrete method has a method body, usually with at least one statement.

This means it has operational code, that gets executed, under the right conditions.

A concrete method is said to **implement** an abstract method, if it overrides one.

Abstract classes and interfaces, can have a mix of abstract and concrete methods.

Method Modifiers

I've already covered access modifiers and what they mean for types, as well as members of types.

And we know we have public, protected, package, and private access modifiers, as options for the members.

Method Modifiers

In addition to access modifiers, methods have other modifiers, which we'll list here, as a high-level introduction.

Modifier	**Purpose**
abstract	When you declare a method abstract, a method body is always omitted. An abstract method can only be declared on an abstract class or an interface.
static	Sometimes called a class method, rather than an instance method, because it's called directly on the Class instance.
final	A method that is final cannot be overridden by subclasses.
default	This modifier is only applicable to an interface, and we'll talk about it in our interface videos.
native	This is another method with no body, but it's very different from the abstract modifier. The method body will be implemented in platform-dependent code, typically written in another programming language such as C. This is an advanced topic and not generally commonly used, and we won't be covering it in this course.
synchronized	This modifier manages how multiple threads will access the code in this method. We'll cover this in a later section on multi-threaded code.